March 1985

Report No. STAN-CS-86-1110

BEST AVAILABLE COPY

86-0337

OK - DTIC

AD-A221 991

TABLOG:
A New Approach to Logic Programming

by

Yonathan Malachi

Zohar Manna

Richard Waldinger

DTIC
ELECTE
MAY 22 1990
S D
D CS D

Department of Computer Science

Stanford University
Stanford, CA 94305

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

90 05 21 088

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) TABLOG: A New Approach to Logic Programming		5. TYPE OF REPORT & PERIOD COVERED technical 6. PERFORMING ORG. REPORT NUMBER STAN-CS-86-1110
7. AUTHOR(s) Yonathan Malachi, Zohar Manna and Richard Waldinger		8. CONTRACT OR GRANT NUMBER(s) N00039-84-C-02111
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Project Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE March 1985 13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

TABLOG [12] is a programming language based on first-order predicate logic with equality that combines relational and functional programming. In addition to featuring both the advantages of functional notation and the power of unification as a binding mechanism, TABLOG also supports a more general subset of standard first-order logic than PROLOG and most other logic-programming languages.

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

The Manna-Waldinger *deductive-tableau* [13,14] proof system is employed as an interpreter for TABLOG in the same way that PROLOG uses a resolution proof system. Unification is used by TABLOG to match a query with a line in the program and to bind arguments. The basic rules of deduction used for computing are a *nonclausal resolution rule* that generalizes classical resolution to arbitrary first-order sentences and an equality rule that is a generalization of narrowing and paramodulation.

In this article we describe the basic features of TABLOG and its (implemented) sequential interpreter, and we discuss some of its properties. We give examples to demonstrate when TABLOG is better than a functional language like LISP and when it is better than a relational language like PROLOG.

March 1985

TABLOG:
A New Approach to Logic Programming

Yonathan Malachi
Zohar Manna

Computer Science Department
Stanford University

Richard Waldinger

Artificial Intelligence Center
SRI International

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

This article appears in *Logic Programming: Relations, Functions, and Equations*, D. Degroot and G. Lindstrom (Editors), Prentice-Hall, 1985. This report supersedes Report Number STAN-CS-84-1012, "TABLOG: the deductive-tablea programming language."

TABLOG:
A New Approach to Logic Programming

Yonathan Malachi
Zohar Manna

Computer Science Department
Stanford University

Richard Waldinger

Artificial Intelligence Center
SRI International

Abstract

TABLOG [12] is a programming language based on first-order predicate logic with equality that combines relational and functional programming. In addition to featuring both the advantages of functional notation and the power of unification as a binding mechanism, TABLOG also supports a more general subset of standard first-order logic than PROLOG and most other logic-programming languages.

The Manna-Waldinger *deductive-tableau* [13,14] proof system is employed as an interpreter for TABLOG in the same way that PROLOG uses a resolution proof system. Unification is used by TABLOG to match a query with a line in the program and to bind arguments. The basic rules of deduction used for computing are a nonclausal resolution rule that generalizes classical resolution to arbitrary first-order sentences and an equality rule that is a generalization of narrowing and paramodulation.

In this article we describe the basic features of TABLOG and its (implemented) sequential interpreter, and we discuss some of its properties. We give examples to demonstrate when TABLOG is better than a functional language like LISP and when it is better than a relational language like PROLOG.

1. Introduction

Logic programming [8] attempts to improve programmer productivity by using predicate logic, a human-oriented language, as a programming language. PROLOG, the most widely

This article appears in *Logic Programming: Relations, Functions, and Equations*. D. Degroot and G. Lindstrom (Editors), Prentice-Hall, 1985. This report supersedes Report Number STAN-CS-84-1012, "TABLOG: the deductive-tablea programming language."

known logic-programming language is based on a resolution proof system and has a restricted syntax. In TABLOG we take the view (shared by other works described in this volume) that logic programming is not to be regarded as synonymous with PROLOG.

A TABLOG program is a list of *assertions* in (quantifier-free) first-order logic with equality that allows one to mix freely functional and relational styles of programming. The use of this richer and more flexible syntax overcomes some of the shortcomings of PROLOG's syntax and makes programming in TABLOG a more intuitive process. For instance, many examples introduced in [15] as mathematical definitions of functions and predicates can be directly executed as TABLOG programs.

The procedural (proof theory) semantics of TABLOG is based on the *deductive-tableau* proof system [13,14]. This powerful proof system, which is described in Section 6, can be applied to arbitrary quantifier-free first-order sentences and therefore does not require the conversion of logic statements into a restricted special form (such as Horn clauses). The execution of a program corresponds to the proof of a *goal*, which produces the desired output(s) as a side effect. When the proof system is used as an interpreter for programs in the TABLOG language, the theorem prover is restrained. Since a particular algorithm is specified by the programmer and since the proof taking place is always a proof of a special case of a theorem—the case for the given input—the program interpreter does not need the full set of deduction rules available in the original deductive-tableau proof system. The theorem prover can thus be more directed, efficient, and predictable than a theorem prover used for program synthesis or for any other general-purpose deduction. When the theorem prover is restricted in order to get these properties, we unfortunately lose the completeness that the general framework enjoys; this fact is briefly treated in the discussion section at the end of the article.

We will first define the syntax of TABLOG and give its flavor using a few simple examples. Before going into more details of TABLOG we will contrast the language with PROLOG and LISP. Later, after giving the proof-theory background in Section 6, we will describe the semantics and execution of TABLOG programs. We will close the article with a brief comparison to other works and a general discussion of our approach. More detailed exposition of TABLOG and its properties can be found in [11].

2. TABLOG Syntax

Syntactic Objects

TABLOG uses the language of the quantifier-free first-order predicate logic with equality. The basic building blocks are:

- truth values: *true, false.*
- connectives: $\wedge$, $\vee$, $\neg$, $\equiv$, $\rightarrow$ (*implies*), $\leftarrow$ (*if*), *if-then-else.*
- variables such as u, v, x_1, y_{25}.
- constants such as a, b, [], 5.
- predicates such as $=$, **prime**, $\in$, $\geq$.
- functions such as **gcd**, **append**, $+$.

We assume the standard definition in logic of *terms* and *formulas* (atomic and compound); an *expression* is either a term or a formula. We do, however, extend these definitions by using the *if-then-else* construct, both as a connective for formulas, e.g.,

$$\text{if } u = [\,] \text{ then } \mathbf{empty}(u) \text{ else } \mathbf{sorted}(u)$$

and as an operator generating terms, e.g.,

$$\mathbf{gcd}(x,\, y) = \text{if } x \geq y \text{ then } \mathbf{gcd}(x-y,\, y) \text{ else } \mathbf{gcd}(x,\, y-x).$$

In the rest of this article we use the term *logic* to refer to the quantifier-free first-order predicate logic extended as above, unless explicitly stated otherwise.

Programs

A program is a list of *assertions* (formulas in the language), specifying an algorithm. Variables are implicitly universally quantified.

Here is a very simple program, for appending two lists:

$$\mathbf{append}([\,], v) \;=\; v.$$
$$\mathbf{append}(x \circ u, v) \;=\; x \circ \mathbf{append}(u, v).$$

The $\circ$ symbol denotes the list insertion operator (**cons** in LISP), $[\,]$ denotes the empty list (**nil** in LISP), and **append** is a function symbol whose semantics is defined by this program.

The inclusion in the syntax of the *if-then-else* construct together with $\leftarrow$ (reverse implication) enables the programmer to write LISP-style as well as PROLOG-style programs.

A call to a program is a *goal* (or query) to be proved. Like the assertions, goals are formulas in logic, but variables are implicitly quantified existentially. The bindings of these variables are recorded throughout the proof and become the outputs of the program upon termination.

For example, a call to the **append** program above might be

$$z \;=\; \mathbf{append}([1,2,3],\; [3,2]).$$

The result of the execution of this program call will be binding z to the output

$$[1,2,3,3,2].$$

The list construct e.g., $[1,2,3]$, is used for convenience in expressing input and output, and is actually an abbreviation for $1 \circ (2 \circ (3 \circ [\,]))$.

3. Examples

The following examples demonstrate a style of programming in TABLOG. The correctness of these programs does not depend on the order of assertions in the program. It is possible, however, to write programs that do take advantage of the known order of the interpreter's goal evaluation, as will be explained later.

In principle, most of the examples described in the other articles in this volume could be executed in TABLOG when written in the appropriate syntax. In particular any PROLOG program that does not use *cut* and does not depend on negation as failure can be trivially converted into an equivalent program directly executable by a TABLOG interpreter.

In the examples, we use x and y (possibly with subscripts) for variables intended to be assigned atoms (integers in most of the examples); u and v (possibly with subscripts) are variables used for lists.

Although we have not as yet described the semantics of TABLOG programs, these examples can be intuitively understood based on the standard meaning of predicate logic.

Deleting a List Element

The following program deletes all (top-level) occurrences of an element x from a list:

$$\begin{aligned}&\textbf{delete}(x, [\,]) \;=\; [\,].\\ &\textbf{delete}(x,\, y \circ u) \;=\; (\text{if } x = y \text{ then } \textbf{delete}(x, u)\\ &\qquad\qquad\qquad\qquad\qquad \text{else } y \circ \textbf{delete}(x, u)).\end{aligned}$$

This program demonstrates the use of equality, *if-then-else*, and recursive calls. If one does not like the explicit conditional *if-then-else*, the last equality assertion can be replaced by the two assertions:

$$\textbf{delete}(x,\, x \circ u) = \textbf{delete}(x, u).$$
$$x \neq y \;\rightarrow\; \textbf{delete}(x,\, y \circ u) = y \circ \textbf{delete}(x, u).$$

To remove all occurrences of a from the list $[a, b, a, c]$, the goal

$$z = \textbf{delete}(a,\, [a, b, a, c])$$

is given to the interpreter.

Quicksort

Here is a TABLOG program that uses the quicksort algorithm to sort a list of numbers. It combines a PROLOG-style relational subprogram for partitioning with a LISP-style functional subprogram for sorting.

1. $\textbf{qsort}([\,]) = [\,].$
2. $\textbf{qsort}(x \circ u) = \textbf{append}(\textbf{qsort}(u_1),\, x \circ \textbf{qsort}(u_2))$
   $\leftarrow\; \textbf{partition}(x, u, u_1, u_2).$
3. $\textbf{partition}(x, [\,], [\,], [\,]).$
4. $\textbf{partition}(x, y \circ u, y \circ u_1, u_2)$
   $\leftarrow\; y \leq x \;\wedge\; \textbf{partition}(x, u, u_1, u_2).$
5. $\textbf{partition}(x, y \circ u, u_1, y \circ u_2)$
   $\leftarrow\; x < y \;\wedge\; \textbf{partition}(x, u, u_1, u_2).$

The assertions in lines 1 and 2 form the sorting subprogram. Line 1 asserts that the empty list is already sorted. Line 2 specifies that, to sort a list $x \circ u$, with head x and tail u, one should append the sorted version of two sublists of u, u_1 and u_2, and insert the element x between them; the subprogram **partition** generates these two sublists, u_1 and u_2, by collecting the elements of u less than or equal to x and greater than x, respectively.

The assertions in lines 3 to 5 specify how to partition a list according to a partitioning element x. Line 3 deals with the partitioning of the empty list, while lines 4 and 5 treat the case in which the list is of the form $y \circ u$. Line 4 is for the case in which y, the head of the list, is less than or equal to x; therefore, y should be inserted into the list u_1 of elements not greater than x. Line 5 is for the opposite case.

The **append** function for appending two lists was defined earlier.

4. Comparison with PROLOG

Functions and Equality

While PROLOG programs must be defined as relations, TABLOG programs can be either relations or functions. The availability of functions and equality makes it possible to write programs more naturally. The functional programming style frees the programmer from needing to introduce many auxiliary variables.

We can compare the PROLOG and TABLOG programs for quicksort. In TABLOG, the program uses the unary function **qsort** to produce a value, whereas a corresponding PROLOG program is defined as a binary relation **qsortp**, in which the second argument is needed to hold the output.

The second assertion in the TABLOG program is

$$\begin{aligned}&\mathbf{qsort}(x \circ u) = \mathbf{append}(\mathbf{qsort}(u_1), x \circ \mathbf{qsort}(u_2))\\ &\qquad \leftarrow \mathbf{partition}(x, u, u_1, u_2).\end{aligned}$$

The corresponding clause in the PROLOG program would be something like

$$\begin{aligned}\mathbf{qsortp}(x \circ u, z) \leftarrow\ &\mathbf{partition}(x, u, u_1, u_2) \wedge\\ &\mathbf{qsortp}(u_1, z_1) \wedge\\ &\mathbf{qsortp}(u_2, z_2) \wedge\\ &\mathbf{appendp}(z_1, x \circ z_2, z).\end{aligned}$$

The additional variables z_1 and z_2 are required to store the results of sorting u_1 and u_2. This demonstrates the advantage of having functions and equality in the language. Note that, although function symbols exist in PROLOG, they are used only for constructing data structures (like TABLOG's primitive functions) and are not reduced.

Since TABLOG includes all the features of (pure) PROLOG, logic-programming techniques like the *difference list* notation or multi-mode (backward) use of relational programs are immediately available in TABLOG.

Recently there have been attempts to add equality to PROLOG. Some of these proposals are described in section 8 on related research, while others are described in other articles in this volume.

Negation and Equivalence

In PROLOG, negation is not available directly; it is simulated by finite failure. To prove $not(P)$, PROLOG attempts to prove P; $not(P)$ succeeds if and only if the proof of P fails. In TABLOG, negation is treated like any other connective of logic. Therefore, we can directly solve queries such as $\neg\mathbf{member}(1, [2, 3])$.

The TABLOG **union** program uses both equivalence and negation:

$$\begin{aligned}&\mathbf{union}([\,], v) = v.\\ &\mathbf{union}(x \circ u, v) = \text{ if } \mathbf{member}(x, v)\\ &\qquad\qquad\qquad\qquad \text{then } \mathbf{union}(u, v)\\ &\qquad\qquad\qquad\qquad \text{else } (x \circ \mathbf{union}(u, v)).\end{aligned}$$

$\neg$**member**$(x, [\,])$.

member$(x, y \circ u) \equiv (x = y) \vee$ **member**(x, u).

Here is a possible PROLOG program of the same algorithm:

unionp$(x \circ u, v, z) \leftarrow$ **memberp**$(x, v) \wedge$ **unionp**(u, v, z).

unionp$(x \circ u, v, x \circ z) \leftarrow$ **unionp**(u, v, z).

unionp$([\,], v, v)$.

memberp$(x, x \circ u)$.

memberp$(x, y \circ u) \leftarrow$ **memberp**(x, u).

Changing the order of the first two clauses in the PROLOG program will result in an incorrect output; the second clause is correct only for the case in which x is not a member of v. The TABLOG assertions can be freely rearranged; this suggests that all of them can be matched against the current goal in parallel, if desired. If in the PROLOG example we add the condition **not**(**memberp**(x, v) as the first conjunct in the body of the second clause, the program will become less order-sensitive but then in many cases **memberp**(x, v) will be evaluated twice, once for the first clause and once for the second.

The Alpine Club Puzzle

The next example shows that some problems are very hard to encode (and solve) when we restrict ourselves to the language of Horn clauses. In addition to negation and equivalence this example also utilizes disjunction.

The following puzzle was the subject of discussion by a few contributors to the PROLOG (electronic) mailing list.

> Tony, Mike and John belong to the Alpine Club. Every member of the Alpine Club is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Mike dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow.
>
> Is there a member of the Alpine Club who is a mountain climber but not a skier?

One of the solutions in PROLOG was offered by R. O'Keefe:

alpinist(Tony).

alpinist(Mike).

alpinist(John).

likes(Tony, rain, yes).

likes(Tony, snow, yes).

likes(Mike, x, yes) $\leftarrow$ **likes**(Tony, x, no).

likes(Mike, x, no) $\leftarrow$ **likes**(Tony, x, yes).

likes(x, rain, no) $\leftarrow$ **climber**(x).

$$\textbf{nonskier}(x) \leftarrow \textbf{likes}(x, \text{snow}).$$
$$\textbf{climber}(x) \leftarrow \textbf{alpinist}(x) \wedge \textbf{nonskier}(x).$$

To solve the puzzle in this form the query

$$\textbf{alpinist}(x) \wedge \textbf{climber}(x) \wedge \textbf{nonskier}(x).$$

should be given to the PROLOG system.

To make sure that the solution is consistent with the original statement of the puzzle we also have to independently show the unprovability of the query

$$\textbf{likes}(x, y, \text{yes}) \wedge \textbf{likes}(x, y, \text{no}).$$

In principle we should also assert in the program

$$\textbf{likes}(x, y, \text{yes}) \vee \textbf{likes}(x, y, \text{no}).$$

However, since the predicate **likes** does not appear in the program or the goal with an uninstantiated third argument, this assertion can be omitted.

Other solutions in PROLOG proposed in the discussion were even less satisfactory as they did not encode the puzzle accurately.

The syntax of TABLOG makes solving this puzzle much more straightforward:

1. $\textbf{alpinist}(\text{John}) \wedge \textbf{alpinist}(\text{Tony}) \wedge \textbf{alpinist}(\text{Mike})$.
2. $\textbf{skier}(x) \vee \textbf{climber}(x) \leftarrow \textbf{alpinist}(x)$.
3. $\neg\textbf{climber}(x) \leftarrow \textbf{likes}(x, \text{rain})$.
4. $\neg\textbf{skier}(x) \leftarrow \neg\textbf{likes}(x, \text{snow})$.
5. $\textbf{likes}(\text{Tony}, \text{rain}) \wedge \textbf{likes}(\text{Tony}, \text{snow})$.
6. $\textbf{likes}(\text{Mike}, y) \equiv \neg\textbf{likes}(\text{Tony}, y)$.

The puzzle is solved by proving the goal

$$\textbf{alpinist}(z) \wedge \textbf{climber}(z) \wedge \neg\textbf{skier}(z).$$

The solution produced by the interpreter is $z = \text{Mike}$.

Note that in line 4 of the program, the procedural interpretation of TABLOG forces us to use the contrapositive

$$\neg\textbf{skier}(x) \leftarrow \neg\textbf{likes}(x, \text{snow})$$

rather than the direct form

$$\textbf{skier}(x) \rightarrow \textbf{likes}(x, \text{snow})$$

which is regarded as a definition for **likes**. This is the only transformation applied to the original specification.

Occur Check

The unification procedure customarily built into PROLOG is not really unification (e.g., as defined in [22]); it does not fail in matching an expression against one of its proper subexpressions since it lacks an *occur check*. When a theorem prover is used as a program interpreter, the omission of the occur check makes it possible to generate cyclic expressions that may not correspond to any concrete objects (and might take infinite amount of time to print).

For example, look at the following program specifying the **parent** relation:

$$\mathbf{parent}(\mathbf{father}(x), x).$$

If this program is called with the goal

$$\mathbf{parent}(z, z)$$

a PROLOG interpreter will succeed but with the binding

$$\{z \leftarrow \mathbf{father}(z)\}$$

i.e.,

$$\{z \leftarrow \mathbf{father}(\mathbf{father}(\mathbf{father}(\cdots)\cdots))\}$$

which is cyclic and cannot be printed unless a special notation for such cases is introduced. This answer is also wrong because logically the program does not imply the truth of the goal. The fact that everyone's father is his or her parent does not imply that someone is his or her own parent.

The unification used by the TABLOG interpreter does include an occur check, so that only theorems can be proved. This choice is orthogonal to the other design decisions in the implementation of TABLOG; if future implementors think that the cost of this test is too high they will be able to use unification without the occur check and pay by losing soundness for some cases. TABLOG allows using nested function calls, and hence programs tend to have fewer repetition of variables than the corresponding PROLOG programs; since the occur check is necessary only if there is at least one variable that occurs more than once in one of the unified expressions (assuming renaming of variables to preserve their locality) this observation can lead to a more efficient unification with restricted application of the occur check.

In the QUTE language [23], the omission of the occur check is essential to the way recursive definitions are introduced. Since QUTE is not based on resolution theorem proving, this does not compromise its soundness.

5. Comparison with LISP

LISP programs are functions, each returning one value; the arguments of a function must be bound before the function is called. In TABLOG, on the other hand, programs can be

either relations or functions, and the arguments need not be bound; these arguments will later be bound by unification.

We can illustrate this with the quicksort program again, concentrating on the partition subprogram. In TABLOG, we have seen how to achieve the partition by a predicate with four arguments, two for input and two for output:

$$\textbf{partition}(x, [\,], [\,], [\,]).$$

$$\textbf{partition}(x, y \circ u, y \circ u_1, u_2)$$
$$\leftarrow\ y \leq x\ \wedge\ \textbf{partition}(x, u, u_1, u_2).$$

$$\textbf{partition}(x, y \circ u, u_1, y \circ u_2)$$
$$\leftarrow\ x < y\ \wedge\ \textbf{partition}(x, u, u_1, u_2).$$

The definition of the program **partition** is much shorter and cleaner than the corresponding LISP program:

$$\textbf{highpart}(x, u)\ \Leftarrow$$
$$\text{if } \textbf{null}(u) \text{ then } \textbf{nil}$$
$$\text{else if } x \geq \textbf{car}(u) \text{ then } \textbf{highpart}(x, \textbf{cdr}(u))$$
$$\text{else } \textbf{cons}(\textbf{car}(u), \textbf{highpart}(x, \textbf{cdr}(u)))$$

$$\textbf{lowpart}(x, u)\ \Leftarrow$$
$$\text{if } \textbf{null}(u) \text{ then } \textbf{nil}$$
$$\text{else if } x \geq \textbf{car}(u)$$
$$\text{then } \textbf{cons}(\textbf{car}(u), \textbf{lowpart}(x, \textbf{cdr}(u)))$$
$$\text{else } \textbf{lowpart}(x, \textbf{cdr}(u)).$$

We can generate the two sublists in LISP simultaneously, but this will require even more pairing and decomposition. Modern LISP systems include provisions for functions with more than one output. Although the syntax for using this feature is somewhat complex, we can get a nicer solution to this problem using multi-valued functions.

Note that unification also gives us "free" decomposition of the list argument into its head and tail; in the LISP program, even if multi-valued functions are used, this decomposition requires explicit calls to the functions **car** and **cdr**. This feature is available as a syntactic sugar in some modern functional languages like SASL [26] and ML [17].

Unification is even more powerful than is indicated by this example. For example, we can generate partially computed results by using a logical variable before it is evaluated, with the concrete value later communicated by the unification. This feature is nicely demonstrated by Reddy's address translation example [19] but it is not limited to relational programs and can be used in a functional language with unification.

6. The Deductive-Tableau Proof System

The deductive-tableau proof system [13,14] is a general framework for theorem proving that was originally utilized for program synthesis. LISP and PROLOG implementations of it

as a part of interactive systems are described in [10], [1], and [27]. Stickel [24] combines the nonclausal resolution rule of this proof system with connection graphs to yield an automatic theorem prover that has been incorporated into a natural-language understanding system.

In this section, we describe the version of the proof system that is used as the TABLOG interpreter; only the deduction rules actually employed by this interpreter are detailed. For a better description of the theory of the general proof system please refer to the original articles. For even more details await [16].

A *deductive tableau* consists of a set of rows, each containing either an *assertion* or a *goal*. The assertions and goals (both of which we refer to by the generic name *entries*) are first-order logic formulas. The tableau is valid if and only if under every interpretation some instance of at least one of the assertions is false or some instance of at least one of the goals is true.

To prove a theorem we enter it as the initial goal; if we want to prove that a sentence is implied by some assumptions we enter the assumptions as assertions and the implied sentence as the goal. In most cases the assertions will specify properties of the data domain. In contrast to standard resolution proof system we do not have to manipulate the negation of the given theorem (using a refutation procedure) and we do not have to convert any sentence to clausal form.

A proof in this system is constructed by adding new goals to the tableau, using deduction rules, in such a way that the final tableau is semantically equivalent to the original one. In the version of the tableau used for the TABLOG interpreter, we follow an affirmation procedure and the proof is complete when we have generated the goal *true*. In general a proof within the tableau can also succeed by refutation (i.e., producing the assertion *false*) rather than by affirmation. Standard resolution and Stickel's implementation of nonclausal resolution use a refutation procedure.

Logically there is duality between assertions and goals: an assertion can be replaced by a goal containing its negation and *vice versa*. By using both assertions and goals we can preserve the intuitive meaning of the sentences.

Deduction Rules

As mentioned before, not all the deduction rules supported by the deductive-tableau proof system are used in the interpreter for TABLOG. The soundness of the inference rules can be justified by case analysis, showing that introducing a new goal in each case will not make the tableau valid unless it was valid before the addition. The basic rules used for the program execution task are the following:

- *Nonclausal Resolution*: This generalized resolution rule allows removal of a subformula P from a goal $\mathcal{G}[P]$ by means of an appropriate assertion $\mathcal{A}[\hat{P}]$. (Note that the use of square brackets here is in the metalanguage and is not related to the list notation). Resolving the goal

  $$\mathcal{G}[P]$$

  with the assertion

  $$\mathcal{A}[\hat{P}],$$

provided that P and $\hat{P}$ are unifiable, i.e., $P\theta = \hat{P}\theta$ for some (most-general) unifier θ, we get the new goal

$$\neg\mathcal{A}'[\mathit{false}] \wedge \mathcal{G}'[\mathit{true}],$$

where $\mathcal{A}'[\mathit{false}]$ is $\mathcal{A}\theta$ after all occurrences of $P\theta$ have been replaced by *false*, and similarly for $\mathcal{G}'[\mathit{true}]$.

This form of the rule is called *goal-assertion resolution*; another form used in TABLOG is the *assertion-goal resolution* that changes the role of the subformulas replaced by *false* and *true*. For example, for the assertion and goal above, assertion-goal resolution will generate the new goal

$$\neg\mathcal{A}'[\mathit{true}] \wedge \mathcal{G}'[\mathit{false}].$$

The choice of the unified subformulas is governed by the *polarity strategy*. A subformula has *positive* polarity if it occurs within an even number of (explicit or implicit) negations, and has *negative* polarity if it occurs within an odd number of negations. Assertions are positive and because of duality every goal has an implicit negation applied to it. A subformula can occur both positively and negatively in a formula. According to the polarity strategy, the instance $\hat{P}\theta$ of the subformula $\hat{P}$ will be replaced by *false* only if $\hat{P}$ occurs with positive polarity; dually, (the instance $P\theta$ of) the subformula P will be replaced by *true* only if P occurs with negative polarity.

Murray [18] proves that nonclausal resolution system is complete for first order logic even under the restriction of the polarity strategy. Note that the version used by TABLOG is not complete because we do not use the versions of the rule that match two assertions or two goals. The version used here always unifies a pair of subformulas while the general rule allows unifying sets of subformulas (and thus takes care of factoring).

- *Equality Rule*: This rule uses an asserted (possibly embedded in a larger formula) equality of two terms to replace one of the terms with the other in a goal. If the asserted equality is conditional, the conditions are added to the resulting goal as conjuncts.

  Thus, suppose the assertion is of the form

  $$\mathcal{A}[s = t],$$

  with the equation $s = t$ occuring in positive polarity, and the goal is

  $$\mathcal{G}[\hat{s}],$$

  where s and $\hat{s}$ are unifiable, i.e., $s\theta = \hat{s}\theta$ for some unifier θ. Then we get the new goal

  $$\neg\mathcal{A}'[\mathit{false}] \wedge \mathcal{G}'[t'],$$

where $\mathcal{A}'[false]$ is $\mathcal{A}\theta$ after all occurrences of the equality $s\theta = t\theta$ have been replaced by *false*, and where $\mathcal{G}'[t']$ is $\mathcal{G}\theta$ after all occurrences of the term $s\theta$ have been replaced by $t\theta$.

The general version of this rule allows matching against the left-hand side or the right-hand side of the equality; in TABLOG however we use this rule in directional way and we always use it to replace the left-hand side by the right-hand side (after the appropriate unification, of course).

- *Equivalence Rule*: This rule replaces one subformula by another asserted to be equivalent to it. This is completely analogous to the equality rule except that we replace atomic formulas rather than terms, using equivalence rather then equality.

While nonclausal resolution and the equivalence rule can be performed unifying arbitrary subformulas, the TABLOG interpreter applies these deduction rules unifying atomic subformulas only.

Each of the above inference rules is followed by *simplification*: a formula is replaced by an equivalent but simpler formula. Both propositional and basic arithmetic simplification are performed automatically by the TABLOG interpreter immediately following every deduction step.

7. Program Semantics

Every line in a program is an assertion in the tableau; a call to the program is a goal in the same tableau. The logical interpretation of a tableau, containing the assertions of a TABLOG program and a goal calling it, is the logical sentence associated with that tableau: the conjunction of the universal closures of the assertions implies the existential closure of the goal.

The desired goal is reduced to *true* by means of the assertions and the deduction rules. The variables are bound when subexpressions of the goal (or derived subgoals) are unified with subexpressions of the assertions. The order of the reduction is explained in the next section. The output of the program is the final binding of the variables of the original goal.

The function symbols of TABLOG are grouped according to their intended use: *constructor* function symbols serve to build data structures in the language; for example, $\circ$ is a predefined constructor. *Basic* (or *built-in* functions have attached procedures hardwired into the implementation to define their semantics; basic arithmetic functions like + and **min** are predefined built-in functions. *Defined functions* are those that are defined by the assertions of a TABLOG program. The constructor and basic functions are called *primitive* functions, while the basic and defined functions are called *reducible* functions. The difference between the two kinds of reducible functions is the way they are reduced: basic functions are reduced by the built-in simplifier while defined functions are reduced by the equality rule. Although there are no constructor predicates we do distinguish between *basic* (primitive) predicates and *defined* predicates. The *primitive operators* include the primitive functions, primitive predicates the logical connectives and the *if-then-else* construct (in both usages). A *primitive expression* is an expression that does not contain defined (i.e., nonprimitive) operators; a *ground expression* is a variable free primitive expression. For example, the term $[(2 + x + 5)]$ (i.e., $(2 + x + 5) \circ [\,]$) is a primitive (but not

ground) expression (with constructor function o, and primitive built-in function +), and will be automatically simplified to $[(x + 7)]$.

As in PROLOG, variables are local to the assertion or goal in which they appear. Renaming of variables is done automatically by the interpreter when there is a collision of names between the goal and assertion involved in a derivation step.

The variables of the original goal are the *output variables*. The interpreter records their bindings throughout the derivation and their final binding is the output of the computation.

8. Program Execution

The tableau system provides deduction rules but does not specify the order in which to apply them. To use this proof system as a programing language, we must devise a *proof procedure* that employs the rules in a predictable and efficient manner.

The proof system is used to execute programs in a way analogous to the inversion of a matrix by linear operations on its rows, where we simultaneously apply the same transformations to the matrix to be inverted and to the identity matrix. In the program execution process, we start with a tableau containing the assertions of the program and a goal calling this program; we apply the same substitutions (obtained by unification) to the current subgoal and to the binding of the output variables. A matrix inversion is complete when we reduce the original matrix to the identity matrix; in TABLOG we are done when we have reduced the original goal to *true*. At this point, the result of the computation is the final binding of the output variables.

Although in the declarative (logical) semantics of the tableau the order of entries is immaterial, the procedural interpretation of the tableau as a program takes this order into account; changing the order of two assertions or changing the order of the conjuncts or disjuncts in an assertion or a goal may lead to different computations and results.

The user must specify an algorithm by employing the predefined order of evaluation of the tableau; the next subsection describes this evaluation order.

Order of Evaluation

At each step of the execution, one *simple expression* (a nonvariable term or an atomic formula) of the current goal is reduced. The expression to be reduced is selected by scanning the goal from left to right. The first (leftmost-outermost) simple expression is chosen and reduced, if possible. The reduction is done by applying an appropriate inference rule: the equality rule for a term, and nonclausal resolution or the equivalence rule for an atomic formula.

If the reduction fails the choice of the simple expression is suspended and a subexpression of it is chosen instead. If no such subexpression exists, a form of backtracking takes place as will be described later.

If the atomic formula is an equality and its two sides do not contain any *defined* functions, the equality is reduced by unifying the two sides and replacing the equality by *true*; if this is not the case or if the unification fails, the choice is suspended and the two sides are searched for the next simple expression.

If the operator (function or predicate) of the chosen expression is primitive it gets special treatment. Operators with built-in semantics (in the form of attached procedures) are evaluated when they have appropriate arguments; otherwise they are treated like failed reductions, i.e., the choice is suspended. Since constructors are not reducible, the choice of a term with a constructor function as the main operator is suspended immediately and subexpressions are reduced.

Formulas generally occur as the outermost expressions; therefore resolution and equivalence rules are in most cases tried first. Only if they cannot be applied do we reduce the terms inside the formulas; this is very similar to the way narrowing is applied in other approaches. Note however that this is not always the case; for example, we can have formulas inside the terms (as the condition of an *if-then-else* expression) and we also have the notion of suspension.

The order of evaluation described here is essentially *lazy evaluation*, as arguments are not computed unless their values are needed. Given the left-to-right order of evaluation between (for example) conjuncts in the goal, we can force the evaluation of an argument by using an auxiliary variable (this is similar to the way [4] removes nested function calls).

Before we demonstrate this with an example, it is important to emphasize again that the matching of the selected expression against program assertions is done in order of appearance. This order dependence makes it possible to guide the control of execution of the program and achieve a more efficient program.

All of the order dependence of programs is part of the sequential model for TABLOG execution. A parallel model does not require programs to be order-dependent.

An Example: Quicksort

Now we will try to illustrate and clarify the description of the last subsection via an instance of a call to the **quicksort** program of section 3.2.

To sort the list $[2,1,4,3]$ using quicksort, we write the goal

$$z = \mathbf{qsort}([2,1,4,3]).$$

Since the right-hand side of the equality contains the defined function **qsort**, the unification of the two sides is delayed and the simple expression chosen for reduction will be the term $\mathbf{qsort}([2,1,4,3])$, i.e., $\mathbf{qsort}(2 \circ [1,4,3])$. This term unifies with the leftmost term $\mathbf{qsort}(x \circ u)$ in the second assertion of the quicksort program,

$$\begin{aligned}\mathbf{qsort}(x \circ u) = \mathbf{append}(&\mathbf{qsort}(u_1), x \circ \mathbf{qsort}(u_2)) \\ \leftarrow\ &\mathbf{partition}(x, u, u_1, u_2).\end{aligned}$$

According to the equality rule, it will be replaced by the corresponding instance of the right-hand side of the equality; this is done only after the unifier

$$\{x \leftarrow 2,\ u \leftarrow [1,4,3]\}$$

is applied to both the goal and the assertion. The occurrence of the equality

$$\mathbf{qsort}(2 \circ [1,4,3]) = \mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2))$$

is replaced by *false* in the (modified) assertion; the occurrence of the term

$$\mathbf{qsort}(2 \circ [1,4,3])$$

is replaced by the term

$$\mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2))$$

in the (modified) goal, and a conjunction is formed, obtaining

$$not(false \leftarrow \mathbf{partition}(2,[1,4,3],u_1,u_2)) \wedge$$
$$z = \mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2)).$$

This formula can be reduced by the simplifications

$$(false \leftarrow P) \Rightarrow not\ P$$

and

$$not(not\ P) \Rightarrow P$$

to obtain the new goal

$$\mathbf{partition}(2,[1,4,3],u_1,u_2) \wedge$$
$$z = \mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2)).$$

Continuing with this example, we now have a case in which the expression to be reduced is an atomic formula, namely,

$$\mathbf{partition}(2,[1,4,3],u_1,u_2).$$

This atomic formula is unifiable with a subformula in the second assertion of the **partition** subprogram (with variables renamed to resolve collisions)

$$\mathbf{partition}(x, y \circ u, y \circ u_3, u_4)$$
$$\leftarrow\ y \leq x\ \wedge\ \mathbf{partition}(x,u,u_3,u_4).$$

Nonclausal resolution is now performed to further reduce the current goal. The unifier

$$\{x \leftarrow 2,\ y \leftarrow 1,\ u \leftarrow [4,3],\ u_1 \leftarrow 1 \circ u_3,\ u_2 \leftarrow u_4\}$$

is applied to both the assertion and the goal; the formula

$$\mathbf{partition}(2,[1,4,3],1 \circ u_3, u_4)$$

is replaced by *false* in the (modified) assertion and by *true* in the goal. Once again a conjunction is formed and the new goal generated (after simplification) is

$$\textbf{partition}(2, [4,3], u_3, u_4) \wedge$$
$$z = \textbf{append}(\textbf{qsort}(1 \circ u_3),\ 2 \circ \textbf{qsort}(u_4)).$$

After a sequence of resolutions to compute the partition of the input list we get the goal

$$z = \textbf{append}(\textbf{qsort}([1]),\ 2 \circ \textbf{qsort}([4,3]))$$

which leads to the selection of the whole right-hand side of the equality as the expression to reduce. None of the two assertions defining **append** can be used to reduce this term; therefore the selection is suspended and the term **qsort**([1]) is chosen instead and gets reduced successfully.

Eventually we reach the subgoal

$$z = [1,2,3,4],$$

where the right-hand side of the equality contains only primitive functions and constants. The execution then terminates and the desired output is

$$[1,2,3,4].$$

Note that some functions and predicates (e.g., $\circ$ in this example) are predefined to be primitive; an expression in which such a symbol is the main operator is never selected to be reduced, although its subexpressions may be reduced.

Backtracking

If the selected expression cannot be reduced, the search for other possible reductions is done by backtracking.

In PROLOG each goal is a conjunction, so all the conjuncts must be proved; this means that, when facing a dead end, we have to undo the most recent binding and try other assertions.

In TABLOG the situation is more complex: each goal (and each assertion) is an arbitrary formula, so it is possible to satisfy it without satisfying all its atomic subformulas. Therefore, when the TABLOG interpreter fails to find an assertion that reduces some basic expression, it tries to reduce the next expression that can allow the proof to proceed. If the expression that cannot be reduced is "essential" (for example, a conjunct in a conjunctive goal), no other subexpression will be attempted and backtracking will occur.

During backtracking, the goal from which the current goal was derived becomes the new current goal, but the next plausible assertion is used. This is similar to the backtracking used in PROLOG.

The Implementation

A prototype interpreter for TABLOG has been implemented in MACLISP. The implemented system serves as a program editor, debugger, and interpreter. All the examples mentioned in this paper have been executed on this interpreter.

The user must declare the variables, constants, functions, and predicates used in the program; some primitive constants, functions, and predicates (such as 0, [], +, −, ≥, **odd**) are predefined.

Because the interpreter is built on top of a versatile theorem-proving system, the execution of programs is relatively slow. The simplifier built into the interpreter now handles complicated cases that might arise in a more general theorem-proving task, but will never occur in TABLOG. We hope that performance will be improved considerably by tuning the simplifier and utilizing tricks from PROLOG implementations to make the binding of variables faster.

9. Related Research

Logic programming has become a fashionable research topic in recent years. Most of the research relates to PROLOG and its extensions. We mention here some of the work that has been done independently of TABLOG to extend the capabilities of PROLOG.

While the deductive-tableau theorem prover used for TABLOG execution is based on a generalized resolution inference rule, [4], [5], and [3] describe a programming language based on a natural-deduction proof system. They do allow quantifiers and other connectives in their language. The language is very general and its execution uses forward as well as backward reasoning. Instead of having the equality rule, all formulas are converted to a *basic form* by eliminating all nested functions. We do not know about the status of implementations for this language. This is the only approach that actually adds all the connectives to logic programming.

Kornfeld [7] extends PROLOG to include equality; asserting equality between two objects in his language causes the system to unify these objects when regular unification fails. This makes it possible to unify objects that differ syntactically. Kornfeld treats only Horn clauses and does not introduce any substitution rule either for equality or for equivalence.

Tamaki [25] extends PROLOG by introducing a reducibility predicate, denoted by ▷. This predicate has semantics similar to the way TABLOG uses equality for rewriting terms. This work also includes *f-symbols* and *d-symbols* that are analogous to TABLOG's distinction between defined and primitive functions. The possible nesting of terms is restricted and programs must be in Horn clause form.

The works of [20], [2], and [9] extend functional programming to have logical variables and unification. By either adding relations or encoding them using functions these languages essentially have most of the power of TABLOG. While EQLOG [2] also contains Horn clauses Reddy's language [20] is essentially functional and [9] takes an even more cautious approach: each function symbol can occur exactly once on the left-hand side of a definition.

There are PROLOG systems, such as LOGLISP [22] and QLOG [6], that are implemented within LISP systems. These systems allow the user to invoke the PROLOG interpreter from within a LISP program and vice versa. In TABLOG, however, LISP-like features and

PROLOG-like features coexist peacefully in the same framework and are processed by the same deductive engine.

10. Discussion

The TABLOG language is a new approach to logic programming: instead of patching up PROLOG with new constructs to eliminate its shortcomings, we suggest a more powerful deductive system.

The combination in TABLOG of unification as a binding mechanism, equality for specifying functions, and first-order logic for specifying predicates creates a rich language that is logically clean. As a consequence, programs more directly correspond to our intuition and are easier to write, read, and modify. We can mix LISP-style and PROLOG-style programming and use whichever is more convenient.

By restricting the general purpose deductive-tableau theorem prover and forcing it to follow a specific search order, we have made it suitable to serve as a program interpreter; the specific search order makes it both more predictable and more efficient than attempting to apply the deduction rules in arbitrary order.

When the theorem prover is restricted to achieve predictability and efficiency, we do lose its completeness. The reasons for losing completeness include: absence of *factoring* or its equivalent in the restricted form of the resolution rule; the omission of *goal-goal* and *assertion-assertion* resolution, and the directionality of using equivalence and equality to define functions and predicates. The absence of completeness does not however affect the examples that we have tried. Furthermore, whenever the programs are restricted to Horn clause form we still enjoy all the completeness properties of this sublanguage.

While the theorem prover supports reasoning with quantified formulas [16, 1], the ramifications of including quantifiers in the language are still under investigation. Quantifiers would certainly enhance the expressive power of TABLOG, but we believe that they are more suited to a specification language than a programming language. If for example we have a universally quantified formula in the condition of an *if-then-else*, in order to evaluate its truth value we have to check the validity of the matrix of the formula. This can be done if we have decision procedures for the data domain of the program, which is reasonable for a general theorem proving system but too expensive for a program interpreter. On the other hand quantifiers may be introduced as an iteration operator. This can be done if we restrict every quantifier to be bounded by some predicate (e.g., membership in a set).

It seems very natural to extend TABLOG to parallel computation. The inclusion of real negation and the conditional *if-then-else* makes it possible to write programs that do not depend on the order of assertions.

The extension of TABLOG to support concurrent programs is being pursued. If the conditions of the assertions are disjoint, several assertions can be matched against the current subgoal in parallel. In addition, disjunctive goals can be split between processes. If there are no common variables, conjuncts can be solved in parallel; otherwise some form of communication is required.

The *or-parallelism* and *and-parallelism* suggested for PROLOG are applicable for TABLOG as well. The or-parallelism of PROLOG corresponds to matching against many assertions;

in TABLOG or-parallelism is also possible within every goal, since, for example, goals can be disjunctive. In TABLOG, other forms of parallelism can be applied to nested function calls.

We believe that TABLOG offers a significant advance over PROLOG in allowing more direct expression of the programmer's intentions. If PROLOG is destined to become the FORTRAN of the year 2000, we can hope that TABLOG will become at least its PASCAL.

Acknowledgments

Thanks are due to Martin Abadi, Yoram Moses, Oren Patashnik, Jon Traugott, and Joe Weening for comments on various versions during the evolution of this paper. We are especially indebted to Bengt Jonsson and Frank Yellin for reading many versions of the manuscript and providing insightful comments and suggestions.

This research was supported in part by the National Science Foundation under grants MCS-82-14523, MCS-81-11586, and MCS-81-05565, by the United States Air Force Office of Scientific Research under Contract AFOSR-81-0014, by the Office of Naval Research under Contract N00014-84-C-0706, and by Defense Advanced Research Projects Agency under Contract N0039-82-C-0250.

The first author gratefully acknowledges the support by an IBM predoctoral fellowship and an HTI fellowship during early stages of this research.

References

1. Bronstein, A. "Full quantification and special relations in a first-order logic theorem prover," unpublished report, Computer Science Department, Stanford University, 1983.

2. Goguen, J.A., and J. Meseguer, "EQLOG: Equality, types, and generic modules for logic programming," in *Logic Programming: Relations, Functions, and Equations*, D. Degroot and G. Lindstrom (Editors), Prentice-Hall, 1985.

3. Hansson, Å., S. Haridi, and S.-Å. Tärnlund, "Properties of a logic programming language," in *Logic Programming*, K.L. Clark and S.-Å. Tärnlund (editors), Academic Press, 1982.

4. Haridi, S. "Logic programming based on a natural deduction system," PhD Thesis, Department of Telecommunication Systems and Computer Science, The Royal Institute of Technology, Stockholm, Sweden, 1981.

5. Haridi, S., and D. Sahlin, "Evaluation of logic programs based on natural deduction," Technical report RITA-CS-8305 B, Department of Telecommunication Systems and Computer Science, The Royal Institute of Technology, Stockholm, Sweden, 1983.

6. Komorowski, H.J. "QLOG: The Programming Environment for PROLOG in LISP," in *Logic Programming*, K.L. Clark and S.-Å. Tärnlund (editors), Academic Press, 1982.

7. Kornfeld, W. "Equality for PROLOG," in *Logic Programming: Relations, Functions, and Equations*, D. Degroot and G. Lindstrom (Editors), Prentice-Hall, 1985.

8. Kowalski, R. *Logic for Problem Solving*, North-Holland, 1979.

9. Lindstrom, G. "Functional Programming and the logical variable," in *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 1985.

10. Malachi, Y. "Deductive programming," unpublished report, Department of Computer Science, Stanford University, December 1982.

11. Malachi, Y. "Nonclausal Logic Programming," PhD Dissertation. Computer Science Department, Stanford University, forthcoming.

12. Malachi, Y., Z. Manna and R. Waldinger, "TABLOG—the deductive-tableau programming language" in *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

13. Manna, Z., and R. Waldinger, "A deductive approach to program synthesis," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, pp. 92–121, January 1980.

14. Manna, Z., and R. Waldinger, "Special relations in automated deduction," Technical Report, No. STAN-CS-85-1051, Department of Computer Science, Stanford University, May 1985. Also, Technical Note 355, Artificial Intelligence Center, SRI International. To appear in *Journal of the ACM.*

15. Manna, Z., and R. Waldinger, *The Logical Basis for Computer Programming, Volume 1: Deductive Reasoning*, Addison-Wesley, 1985.

16. Manna, Z., and R. Waldinger, *The Logical Basis for Computer Programming, Volume 2: Deductive Systems*, Addison-Wesley, to appear.

17. Milner, R. "A proposal for standard ML," *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

18. Murray, N.V. "Completely nonclausal theorem proving," *Artificial Intelligence*, Vol. 18, No. 1, pp. 67–85, 1982.

19. Reddy, U. "On the relationship between logic and functional languages," in *Logic Programming: Relations, Functions, and Equations*, D. Degroot and G. Lindstrom (Editors), Prentice-Hall, 1985.

20. Reddy, U. "Narrowing as the operational semantics of functional languages," in *Proceedings of the 1985 Symposium on Logic Programming*, Boston, Massachusetts, July 15–18, 1985.

21. Robinson, J.A. "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, Vol. 12, No. 1, January 1965, pp. 23–41.

22. Robinson, J.A., and E. E. Sibert, "LOGLISP: An alternative to PROLOG," in *Machine Intelligence 10*, J. E. Hayes, D. Michie, and Y-H Pao (editors), Ellis Horwood Ltd., Chichester, 1982.

23. Sato, M., and T. Sakurai, "QUTE: A Functional language based on unification," in *Logic Programming: Relations, Functions, and Equations*, D. Degroot and G. Lindstrom (Editors), Prentice-Hall, 1985.

24. Stickel, M.E. "A nonclausal connection-graph resolution theorem-proving program," *Proceedings of the National Conference on Artificial intelligence*, Pittsburgh, Pennsylvania, August 1982.

25. Tamaki, H. "Semantics of a logic programming language with a reducibility predicate," *Proceedings of the IEEE Logic Programming Conference*, Atlantic City, February 1984.

26. Turner, D.A. "SASL language manual," Computer Laboratory, University of Kent, Canterbury, England, 1976 (revised August 1979).

27. Yellin, F. "PROLOG based program synthesis." unpublished report, Computer Science Department, Stanford University, 1983.